Tutorial: LearnOOP 6

LearnOOP is a shape-drawing program that introduces you to the basics of object-oriented programming. Each of the shapes that LearnOOP draws is an object. This tutorial shows how the routines that operate on objects are encapsulated in the class declaration, how dynamic binding and inheritance work, and how to define a new class yourself.

If you already know object-oriented programming, you don't have to read this chapter.

Before you begin

Make sure that you've installed the THINK Project Manager on your Macintosh as described in the instructions in Chapter 2, "Installation."

What you should know

This tutorial assumes that you know how the THINK Project Manager works and how to use the THINK Debugger. You should know how to open a project, how to run a project, and how to use the editor. If you need to learn how to do any of these things, see the tutorial in Chapter 3, "Tutorial: Hello World." If you need to learn how to use the debugger, complete the tutorial "Bullseye" in Chapter 5 before you try LearnOOP.

Contents

What Does LearnOOP Do? 97
How Does LearnOOP Work?. 99
How Do Objects Know How to Draw Themselves? 100
The LearnOOP Class Declarations 104
How Do You Create an Object?. 107
How Do You Define a New Class?. 108
But How Does It Know? 110
Where to Go Next 111

What Does LearnOOP Do?

To begin this tutorial, double-click the `LearnOOP.π` project in the `LearnOOP` folder. If you're already in the THINK Project Manager, close any existing project, and choose `LearnOOP.π` from the standard file dialog. The Project window looks like this:

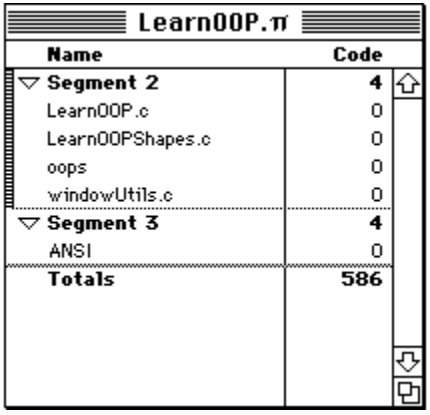

Figure 6-1 LearnOOP Project window

Choose **Run** from the **Project** menu to start the program. The THINK Project Manager loads the libraries and compiles the source files before LearnOOP starts running. It takes a minute or so.

The LearnOOP program uses the console window to prompt you for something to do, and it draws in another window called the Drawing window. When you run the program, LearnOOP asks you if you want to create an Oval or a Rectangle:

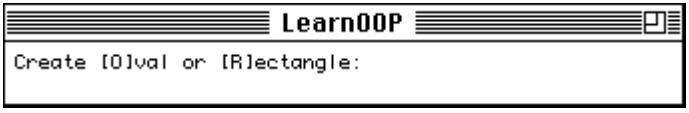

Figure 6-2 LearnOOP asking for a shape

If you want to draw an oval, press O; if you want to draw a rectangle, press R. To quit the program altogether, press Q. Right now, press O, and then the Return key.

After you tell LearnOOP what kind of shape to draw, it asks you to give it the top, left, bottom, and right coordinates of a rectangle that completely encloses the shape:

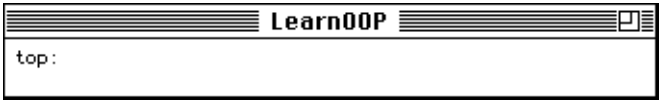

Figure 6-3 LearnOOP asking for a coordinate

The point 0,0 is at the top-left corner of the window. To see how this works, try typing these values: `10`, `10`, `40`, `90`. Press the Return key after each value. LearnOOP draws the shape in the Drawing window like this:

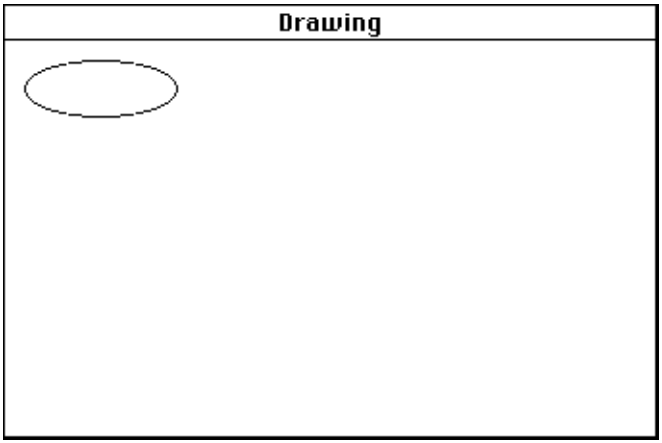

Figure 6-4 Oval with coordinates 10 10 40 90

After it draws the first shape, LearnOOP asks what kind of shape to draw next. Try a few more shapes before you go on to the next section.

How Does LearnOOP Work?

If LearnOOP is still asking you for a kind of shape, press Q to make it stop.

Double-click the file named `LearnOOP.c` in the Project window. The program that draws shapes is pretty simple. Look at `main ()` at the end of the file:

```
/****
 * main
 *
 * Prompt for a shape and its location.
 * Then draw it in the drawing window.
 *
 ****/

void main( void)
{
   CShape   *aShape;
   short    top, left, bottom, right;

   SetUpWindows();

   while ((aShape=ChooseShape()) != NULL) {
      top = GetNum("top");
      left = GetNum("left");
      bottom = GetNum("bottom");
      right = GetNum("right");

      aShape->SetBounds(top, left,
                        bottom, right);
      aShape->Draw();
   }
}
```

`SetUpWindows()` is in the file `windowUtils.c`. If you're curious about how the windows in this tutorial work, take a look at that file.

The first function, `SetUpWindows`, sets up the console and Drawing windows.

The next part of the program in the `while` loop actually does the drawing. The function `ChooseShape` asks you for the kind of shape to draw. If you choose to quit, `ChooseShape` returns `NULL`, and the program ends. Otherwise, `ChooseShape` creates a shape and stores it in the variable `aShape`.

The next four function calls ask you for the coordinates of the shape and store them in four variables.

The next two lines are the most interesting lines of the program. The line

```
aShape->SetBounds(top, left,
                  bottom, right);
```

sends a `SetBounds` message to the shape `aShape` to give it its coordinates. The line

```
aShape->Draw();
```

sends `aShape` a message to draw the shape.

"Sending a message" is object-oriented programming terminology for calling a method. You can think of "sending a message" as calling a function through an object.

The important thing to notice is that the program uses the same two lines to set the coordinates and to draw the shapes regardless of the kind of shape. The statement that draws the shape doesn't need to know what kind of shape it will draw. It just sends the shape object a `Draw` message. The object knows how to draw itself.

How Do Objects Know How to Draw Themselves?

The best way to find out what's going on is to step through the program. If LearnOOP is asking you for a shape, press Q to make the program end.

To learn how to use the source-level debugger, see Chapter 5, "Tutorial: Bullseye," and Chapter 12, "Debugging Programs," in this guide.

Choose **Use Debugger** from the **Project** menu to turn on the source-level debugger. The "bug" column will appear along the left edge of the project window.

Note

Remember that you need to be running System 7 (or MutiFinder under System 6) to use the THINK Project Manager's source-level debugger. If you're running on a Macintosh with only 1 megabyte of RAM and you can't use the debugger, just read along in this section.

Choose **Run** from the **Project** menu to start LearnOOP. Instead of launching LearnOOP, the THINK Project Manager starts the source-

level debugger. Click the diamond next to the line that sends the Draw message. Your debugger window should look like this:

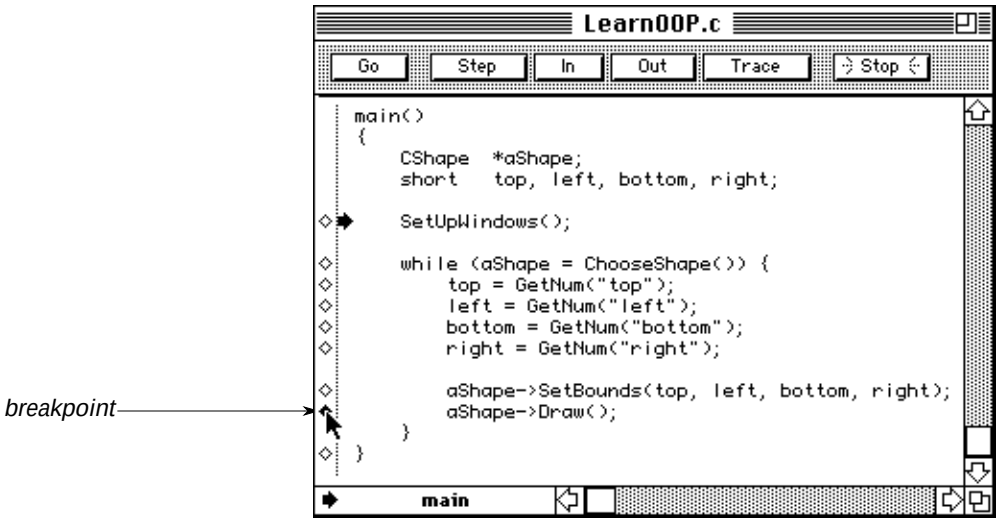

Figure 6-5 A breakpoint before the Draw message

Now click the Go button in the debugger's Source window.

The THINK Project Manager starts the program again, and LearnOOP asks you to choose a shape. Press R for rectangle. LearnOOP asks you to type in the bounds for the shape. Type whatever you like. If you can't think of anything, try `20`, `30`, `45`, `66`. Be sure to press the Return key after each number.

As soon as you press Return after the last coordinate, LearnOOP stops at the line that sends a `Draw` message to `aShape`. To find out what's going on, click the In button in the debugger's Source

window. You see the current statement arrow point to the beginning of a function called `CRectangle::Draw()`:

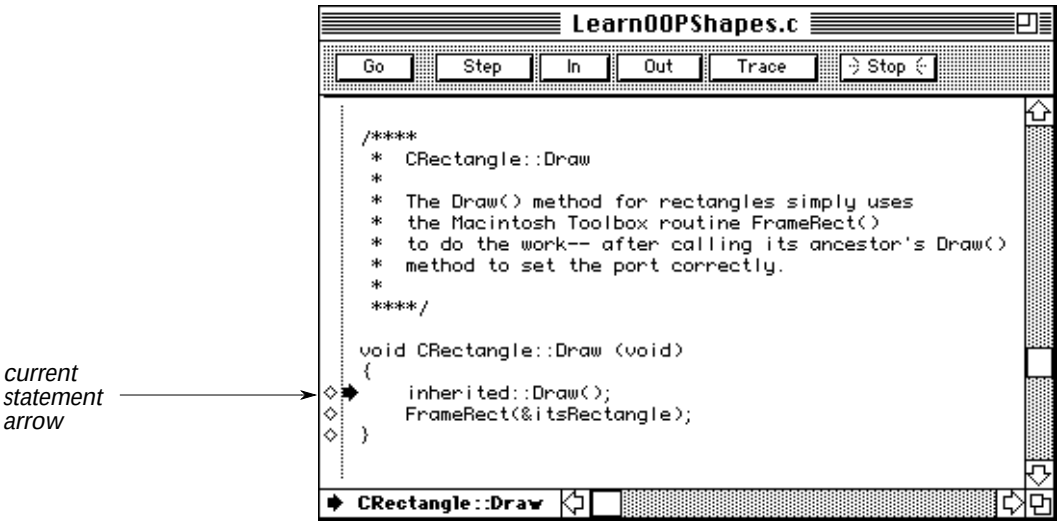

Figure 6-6 Execution stopped in CRectangle::Draw()

This procedure is actually a method. In fact, it's the method that handles `Draw` messages sent to objects of class `CRectangle`. In a moment, you'll see what this class looks like. But first, see what happens when you choose a different kind of shape.

Click the Go button in the debugger's Source window to let LearnOOP draw the rectangle. When it asks you for another shape, press O for oval. Type whatever you want for the location. Because the breakpoint is still set, the program stops at the line that sends a `Draw` message to `aShape`. Click the In button in the debugger's

window again. This time, you see the current statement arrow point to the beginning of another function:

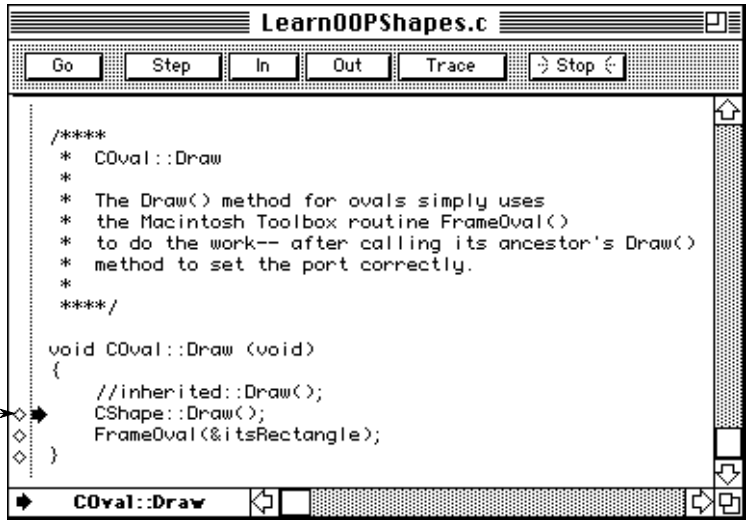

Figure 6-7 Execution stopped in COval::Draw()

The procedure `COval::Draw()` is the method that handles `Draw` messages for objects of class `COval`.

The next section explains how the shape-drawing classes are declared. You can keep exploring with the debugger. When you're ready to go on, press Q to quit the program.

The LearnOOP Class Declarations

The declaration of a class determines how an object responds to a message. The file LearnOOPShapes.h contains the class declarations for the shapes. To open LearnOOPShapes.h, choose the **Open** command from the **File** menu, or hold down the Option key as you click the title bar of the Project window and choose LearnOOPShapes.h from the pop-up menu.

The class declarations in LearnOOPShapes.h look like this:

```
      // CShape is the abstract class that
      // describes all shapes. All shape
      // classes are subclasses of this class.

   #if __cplusplus
   struct CShape {            // Compile with
   #else                      // Symantec C++!
   struct CShape : indirect { // Or THINK C!
   #endif

         Rect  itsRectangle;
                        // The rectangle that
                        // encloses the shape.

      // The SetBounds method sets values
      // for itsRectangle. Most subclasses
      // won't need to override it.

      virtual void SetBounds (short aTop,
                              short aLeft,
                              short aBottom,
                              short aRight);

      // The Draw method draws the shape.
      // All subclasses should override it, so
      // we declare it 'virtual' in this
      // class.

      virtual void Draw(void);
   };

   struct CRectangle : CShape {
      void Draw(void);
   };

   struct COval : CShape {
      void Draw(void);
   };
```

The declarations in this file describe three classes. The first class, `CShape`, is a generic shape. Every object of class `CShape` has one instance variable: a rectangle called `itsRectangle`. Every object of class `CShape` also implements two methods: `SetBounds` to set the size of the shape, and `Draw` to actually draw the shape. These are virtual methods, which means you can override them with new methods when you create subclasses of `CShape`.

`CShape` is an abstract class. You use an abstract class as the common ancestor of a family of classes that share traits and behaviors. You never create an object of an abstract class. Instead you create subclasses of the abstract class.

The common trait that all shapes share is a bounding rectangle. The common behaviors that all shapes share are being drawn and setting the bounding rectangle.

The next two classes, `CRectangle` and `COval`, are subclasses of `CShape`. They both inherit the `itsRectangle` instance variable and the `SetBounds` method. Both classes override the `Draw` method. This means that a `CRectangle` is identical to a generic shape, but it has its own drawing behavior. `COval` is also identical to a generic shape, but it has its own drawing behavior different from `CRectangle`.

A set of related classes is called a class hierarchy. Figure 6-8 shows you the class hierarchy for the LearnOOP program. The diagram shows that `CRectangle` and `COval` are descendants of the class `CShape`.

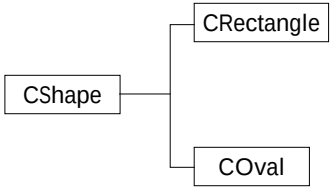

Figure 6-8 The LearnOOP class hierarchy

Now look at the implementation of the methods. The methods are defined in the file `LearnOOPShapes.c`. Use the **Open** command from the **File** menu or double-click the name `LearnOOPShapes.c` in the project window to open the file.

These are the method definitions for the abstract class CShape:

```
void CShape::SetBounds (short aTop,
                        short aLeft,
                        short aBottom,
                        short aRight)
{
   itsRectangle.top = aTop;
   itsRectangle.left = aLeft;
   itsRectangle.bottom = aBottom;
   itsRectangle.right = aRight;
}

void CShape::Draw (void)
{
   SetPort(DrawingWindow);
}
```

The first method, SetBounds, just sets up the boundary rectangle itsRectangle.

The second method is the generic Draw method. Because all shapes in this program appear in the Drawing window, the Draw method sets the drawing port to the window stored in the global variable DrawingWindow. Every subclass of CShape uses CShape's Draw method as well as implementing its own.

Because CShape is an abstract class, there shouldn't be an object of class CShape. Every subclass of CShape should override the Draw method to draw a particular kind of shape.

The Draw methods for CRectangle and COval override the generic Draw method with the specific behavior for their class.

```
void CRectangle::Draw (void)
{
   inherited::Draw();
   FrameRect(&itsRectangle);
}


void COval::Draw (void)
{
   inherited::Draw();
   FrameOval(&itsRectangle);
}
```

These are the two functions you saw when you stepped through the program. They're what give CRectangle and COval their unique

drawing behavior. First they call `inherited::Draw()` to invoke the superclass's `Draw` method to set up the Drawing window. Then they call the appropriate Macintosh Toolbox routine.

If you send an object of class `CRectangle` a `Draw` message, the `CRectangle::Draw()` method is called. First, the method invokes `CShape`'s `Draw` method to set up the Drawing window. Then it calls the Macintosh Toolbox routine `FrameRect` to draw the rectangle. If you send an object of class `COval` a `Draw` message, the `COval::Draw()` method is called. It invokes `CShape`'s `Draw` method; the `FrameOval` procedure draws an oval.

How Do You Create an Object?

The specific `Draw` method that gets called when you send an object a `Draw` message depends on the class of the object. So, how do you create objects of different classes?

Look at the `ChooseShape` function in the file `LearnOOP.c`. This is the function that asks you to choose the shape:

```
CShape *ChooseShape (void)
{
   char  c;

   while (1) {
      c = GetCh("Create [O]val or "
                "[R]ectangle");

      switch (c) {
         case 'Q':
         case 'q':
            return NULL;

         case 'R':
         case 'r':
            return new CRectangle;

         case 'O':
         case 'o':
            return new COval;

         default:    /* stay in the loop */
            break;
      }
   }
}
```

After it asks you for the kind of shape to draw, this function looks at what you pressed and returns an object of the appropriate class.

If you pressed R or O, it uses the operator `new` to create an object. When the THINK C + Objects option on the Language Extensions page of the THINK Project Manager **Options** dialog is on, the word `new` becomes a keyword that creates an object of the named class.

Note that the return type of `ChooseShape` is `CShape*`. The rules for object assignment say that you can assign an object pointer of a subclass to an object pointer of a superclass. What this means is that you can assign a `CRectangle` pointer to a variable of class `CShape*`, but you can't assign a pointer of type `CShape*` to a pointer of type `CRectangle*`.

After it creates the shape, `ChooseShape` returns the new object pointer, and the object gets set to the global variable `aShape` in the `while` loop in `main()`:

```
while (aShape = ChooseShape()) {
```

The type of `aShape` is `CShape*`. Doesn't this mean that when you send a `Draw` message to `aShape`, it should call the generic `CShape::Draw()` method? No, because the class of an object is determined at run-time. In other words, the type of the variable `aShape` is `CShape*`, but the class of the object that it points to is either `CRectangle` or `COval`. The class is determined while your program is running. This is called dynamic binding.

So if you create an object of class `CRectangle` and send it a `Draw` message, the method that gets called is `CRectangle::Draw()`.

What happens when you send the `SetBounds` message? Since neither the `CRectangle` class nor the `COval` class overrides the `SetBounds` method, the generic `CShape::SetBounds()` method gets called when you send either class of object a `SetBounds` message. This is called inheritance. The `CRectangle` and `COval` classes inherit the `SetBounds` method from their superclass, `CShape`.

How Do You Define a New Class?

Now try creating a new class that's a subclass of one of the existing three classes. Here's how to create a subclass to draw squares.

The first thing to do is to decide which class to subclass. Since a square is just like a rectangle—except that it has sides of equal lengths—try making the square class a subclass of `CRectangle`. You can use the same drawing method, but you have to override the `SetBounds` method. Here's how you'd declare the new `CSquare` class:

If you want to follow along, put this class declaration right under the `COval` *declaration in the file* `LearnOOPShapes.h`.

```
struct   CSquare : CRectangle {
   void  SetBounds (short aTop,
                    short aLeft,
                    short aBottom,
                    short aRight);
};
```

And here's one way to define the `SetBounds` method for the `CSquare` class:

If you want to follow along, put this class declaration right under the `COval::Draw()` *definition in the file* `LearnOOPShapes.c`.

```
void CSquare::SetBounds (short aTop,
                         short aLeft,
                         short aBottom,
                         short aRight)
{
   aRight = aLeft + (aBottom - aTop);
   inherited::SetBounds(aTop, aLeft,
                        aBottom, aRight);
}
```

This function says to reset the right edge of the bounding rectangle to the same distance from the left edge as the bottom edge is from the top edge. (It ignores whatever value you give it for `aRight`.) The next line says to use the superclass's `SetBounds` method. In this case, it means to call `CRectangle`'s `SetBounds` method, which it inherits from `CShape`.

The only thing left to do is to edit the `ChooseShape` function in the file `LearnOOP.c` to handle the new shape. Here's how:

```
CShape *ChooseShape (void)
{
   char  c;

   while (1) {
      c = GetCh("Create Oval, Rectangle, "
                " Square");

      switch (c) {
         case 'Q':
         case 'q':
            return NULL;

         case 'R':
         case 'r':
            return new CRectangle;

         case 'O':
         case 'o':
            return new COval;

         case 'S':
         case 's':
            return new CSquare;

         default:    /* stay in the loop */
            break;
      }
   }
}
```

Try running the program now with the new class. When you ask `ChooseShape` to create a square, it should ignore the value you give for the right edge of the bounding rectangle.

Note

Because of the way LearnOOP asks for the coordinates of the bounding rectangle, you still have to provide a value for the right edge.

But How Does It Know?

You may still be wondering what in the LearnOOP program is calling the right method for a given message. There is a mechanism that figures out which method goes with which message for a given object. But this mechanism is abstracted away in the compiler and is part of the language.

Here's an example. You probably know that most microprocessors use different machine-language instructions for multiplying integers and for multiplying floating-point numbers. But when you write a C program, you know that you can write statements like:

```
floatVar = 1961.0102 * floatVal;
count = 24 * intVal;
```

Though there's a "hidden mechanism" that knows when to use floating-point multiplication and when to do integer multiplication, you don't worry about it. The inner workings of the multiplication operator are abstracted away and are part of the language.

It's the same thing with object-oriented programming. The mechanism that matches messages and methods with objects is part of the language. The power of object-oriented programming comes not from how it works but from the layer of abstraction that it adds to the low-level mechanics of data structures and algorithms.

Where to Go Next

Since this small program declares only three classes, this is a good time to learn how to use the Class Browser. Chapter 14, "Browsing Classes," describes the Class Browser in detail.

If you want, you can try adding more methods and classes. Try adding a method that asks where to put the shape so you don't have to provide an extra value for the `CSquare` class. Or you might want to try adding methods for filled shapes or for shapes with different line thicknesses.

6 Tutorial: LearnOOP